Doublelength Floating-Point Arithmetic on the TMS320C30

APPLICATION REPORT: SPRA114

Al Lovrich
Digital Signal Processor Products
Semiconductor Group
Texas Instruments

Digital Signal Processing Solutions

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Doublelength Floating-Point Arithmetic on the TMS320C30

Abstract

This chapter, reprinted from *IEEE Micro Magazine*, describes the third generation of the TMS320 family of digital signal processors, the TMS320C30. It describes the origin and development of the 32-bit floating-point device. The topics covered include:

- An overview of the characteristics of the TMS320C30 processor
- A description of the architecture of the 320C30
- A description of the software features of the programmable DSP
- A description of development tools and support
- The types of demanding applications for which the 320C30 is most suitable

Support graphics include:

- Architecture block diagrams
- Diagrams showing on-chip memory, cache and buses, the 320C30 central processing unit, and peripheral bus and peripherals
- A pipeline of 320C30 instructions
- Sample code implementations

The chapter closes with an endnote about the likely direction of this technology, a list of references and some biographical information about the authors.

Product Support

World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Email

For technical issues or clarification on switching products, please send a detailed email to (dsph@ti.com). Questions receive prompt attention and are usually answered within one business day.

In the past, extended-precision arithmetic has been implemented only on fixed-point processors. The introduction of the TMS320C30 Digital Signal Processor (DSP), a floating-point 33-MFLOP device, enables us to represent multilength floating-point math in terms of singlelength floating-point math. Extended-precision arithmetic allows designers to have more accuracy in their applications. Some of these applications include digital filtering, FFTs, image processing, control, etc.

This application report describes how to extend the available precision of floating-point arithmetic on the TMS320C30. Our emphasis is on implementing an efficient extension of the available precision while minimizing both the execution time and the memory usage.

The structure of this report is as follows: The first section describes the TMS320C30 DSP floating-point number representation. The second section discusses doublelength arithmetic and some basic definitions. The third section discusses the algorithms used along with the TMS320C30 implementation. An analysis of the error introduced by the algorithm is presented in the fourth section. The last section provides an insight into generating C-callable functions from assembly language routines. Finally, the appendix provides the source listings for the extended-precision arithmetic.

Floating Point Format

The TMS320C30 supports three floating-point formats [1].

- Short floating-point format, used to represent immediate operands, consisting of a 4-bit exponent and a 12-bit mantissa.
- Single-precision format, used for regular floating-point value representation, consisting of an 8-bit exponent and a 24-bit mantissa.
- The extended-precision format, used with the extended-precision registers, consisting of an 8-bit exponent and a 32-bit mantissa.

For the extended-precision algorithms to work properly on the DSP, it is important to start from the highest-precision floating-point format available in the system that is used for basic floating-point operations. The single-precision format is of particular interest in developing the TMS320C30 code for extended-precision floating-point operations. Therefore, a working knowledge of the properties of this format is essential for the concepts presented in this application report.

In the single-precision format, the floating-point number is represented by an 8-bit exponent field (e) in two's complement notation, and a two's complement 24-bit mantissa field (f) with an implied most-significant nonsign bit. Bit 23 of the mantissa indicates the sign (s), as shown in Figure 1.

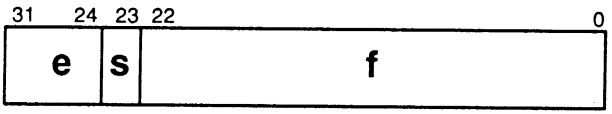

Figure 1. Single-Precision Floating-Point Format of the TMS320C30

Operations are performed with an implied binary point between bits 23 and 22. When the implied most-significant nonsign bit is made explicit, it is located to the immediate left of the binary point after the sign bit. We show the implied bit explicitly throughout this application report for clarity. The floating-point number x is expressed as follows:

$$x = \begin{cases} 01.f \times 2^e & \text{if } s = 0; \\ 10.f \times 2^e & \text{if } s = 1; \\ 0 & \text{if } e = -128,\ s = 0, \text{ and } f = 0 \end{cases}$$

The range and precision available with the TMS320C30 single-precision floating-point format are illustrated by the following values:

Most Positive:	$x = +3.4028234 \times 10^{+38}$
Least Positive:	$x = +5.8774717 \times 10^{-39}$
Least Negative:	$x = -5.8774724 \times 10^{-39}$
Most Negative:	$x = -3.4028236 \times 10^{+38}$

Doublelength Floating-Point – The Basics

The techniques used to develop doublelength results in this application report require a singlelength floating-point system and arithmetic that satisfy certain conditions. The TMS320C30 implementation takes the singlelength system as the highest floating-point precision system available. The algorithms presented do not require a doublelength accumulator with respect to the singlelength system used. The extended-precision formats available are used to control the truncation or rounding of the single-precision results.

The doublelength arithmetic presented here increases precision of a given floating-point operation without the need for a doublelength accumulator. Using this method, the result of the floating-point operations on two single-precision numbers can be determined exactly. If x and y are two such numbers and the desired operation is addition, the result can be represented as a pair of floating-point numbers z and zz. The z value represents

the most significant portion of the floating-point operation, while zz represents the least significant portion of the floating-point operation.

As an example, consider the result of the exact addition of two floating-point numbers x and y that are expressed in the single-precision format of the TMS320C30:

x = 217FFFFFh (decimal: $1.71798682 \times 10^{10}$)
y = 0C7FFFFFh (decimal: 8.19199951×10^{3})

The values are represented in the TMS320C30 binary equivalent as follows:

$x = 2^{33} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1111b$
$y = 2^{12} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1111b$

Addition of two floating-point numbers requires aligning the two variables x and y [1]:

$x = 2^{33} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1111b$
$y = 2^{33} \times 00.000\ 0000\ 0000\ 0000\ 0000\ 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000b$

As can be seen in this example, most of the precision available for y will not be available to carry out the addition. Maintaining full precision for floating-point addition requires extra mantissa bits beyond the 24 bits available on the DSP. Since the need for such precision is rare, software methods are used to represent the result of the operation as a floating-point number pair (z,zz). In our example, the exact result is represented as follows:

$z = 2^{34} \times 01.000\ 0000\ 0000\ 0000\ 0000\ 0011b$
$zz = 2^{09} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1000b$

The corresponding hexadecimal representation of (z,zz) is shown below:

z = 22000003h (decimal: $1.71798753 \times 10^{10}$)
zz = 097FFFF8h (decimal: 1.0239995×10^{3})

Some definitions are basic to the development of concepts in this report. First is the definition of the floating-point operations over a system *R*. The system contains all the possible floating-point numbers that the single-precision format of the TMS320C30 can represent. All the floating-point arithmetic is carried out in base 2. Therefore, *R* can be represented as follows on the TMS320C30:

$$R = \{x | x = m(x)2^{e(x)},\ |m(x)| < 2^{24},\ -128 < e(x) < 127\}$$

A floating-point operation is *faithful* if the result of the operation fl(x *y) equals either:

The largest element of *R* that is smaller than or equal to (x * y) or

The smallest element of *R* that is larger than or equal to (x * y)

where * represents one of the following floating-point operations: +, −, ×, ÷. In other words, faithful refers to truncating the floating-point operation result. The floating-point

multiplier on the TMS320C30 saves the upper 40 bits of the mantissa in one of the extended-precision registers [1] and drops the least significant byte of the result. By this definition, the floating-point multiplication on the TMS320C30 is *faithful*. Since the algorithms require the floating-point result to be in single-precision format, the floating-point multiplication on the DSP must therefore be followed by a second truncation step. Saving the contents of the extended-precision register to a memory location or masking off the low 8 bits results in truncation.

A floating-point operation is *optimal* if for all x and y, the result of fl(x * y) is an element of R nearest to (x * y). In other words, the round-off error should not exceed one-half of the last remaining bit position. This is commonly referred to as *rounding*.

The results of floating-point operations on the TMS320C30 are stored in the extended-precision registers [1]. The extended-precision register adds 8 bits of precision to the floating-point arithmetic result. Execution of the RND (round) instruction forces the result of the floating-point arithmetic to be *optimal*. When you round the result of the addition or subtraction operations on the TMS320C30, these floating-point operations become *optimal*.

Implementing Doublelength Floating-Point Arithmetic

This section presents the algorithms used in implementing doublelength arithmetic in pseudo-code for a number of fundamental floating-point operations. The basic idea of doublelength arithmetic can be extended to multiplelength precision, given that the start of the implementation is based on the highest precision available on the system. Therefore, to achieve quadruplelength results, the same algorithm can be applied to doublelength values, and so on. The implementation is based on the theoretical results presented in Reference [2].

Exact Singlelength Addition

In this discussion of the algorithm used to carry out *exact* addition and its implementation on the TMS320C30 DSP, the term *exact* refers to performing an operation on two floating-point numbers, x and y, and obtaining a doublelength floating-point number pair (z,zz) to represent the result. In this implementation, we have not accounted for floating-point exponent overflow or underflow. For this algorithm to produce a correct result, the floating-point addition and subtraction must be *optimal*.

The purpose of *exact* addition is to find a term, zz, that satisfies Equation (2).

$$z + zz = x + y \tag{2}$$

Equation (2) can be rewritten as

$$zz = y - (z - x) \tag{3}$$

Equation (3) can be expanded into Equation (4).

$$w = z - x$$
$$zz = y - w \tag{4}$$

In particular, $|x| > |y|$ must be valid for Equation (4) to be valid. Implementation of Equation (4) on the TMS320C30 always generates the exact correction term zz if the result of floating-point addition operation is made *optimal*. This requirement guarantees that the result of single-precision floating-point add and subtract belongs to system R. By swapping the x and y values when $|x| < |y|$, the condition for obtaining an *exact* result is met.

The algorithm requires that x and y be normalized. Normalization guarantees that the floating-point number has only one sign bit, and that sign bit is followed by nonsign bits [1]. Floating-point addition on the TMS320C30 assumes that the operands are normalized.

The TMS320C30 assembly code for obtaining the doublelength sum of two singlelength floating-point numbers x and y is shown in Appendix A. First, the values for x and y are interchanged when $|x| < |y|$. When you add x and y values, the number with the smaller exponent, y, is shifted repeatedly until the exponents of x and y are equal and their mantissas are aligned. We have now calculated the singlelength number, z, that satisfies Equation (2). Since the floating-point addition on the TMS320C30 is made optimal by rounding, the extra precision is, in effect, dropped. The extra precision value, zz, is obtained by implementing Equation (4). Figure 2 is a graphical representation of the implemented algorithm. The figure also shows the relationship between doublelength number pair (z,zz) and singlelength floating-point numbers and their representation on the TMS320C30.

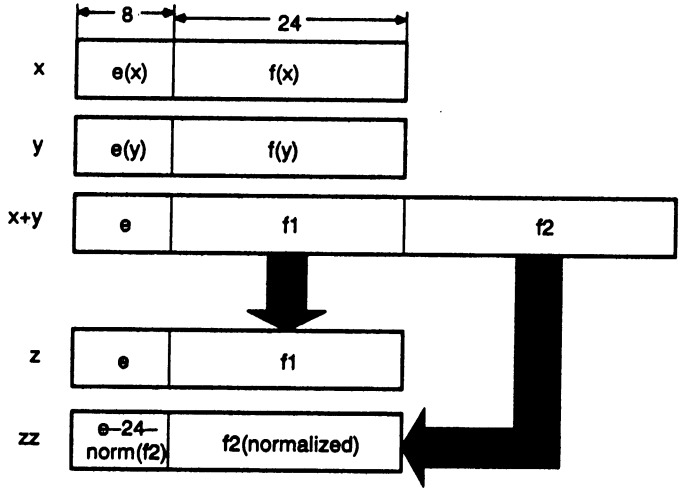

Figure 2. Exact Singlelength Addition

The same algorithm can be used to implement exact floating-point subtraction on the DSP. This is accomplished by negating the second operand and performing an exact addition.

Doublelength Addition

A natural extension of exact singlelength addition and subtraction is its application to doublelength arithmetic. Figure 3 shows an algorithm for implementing doublelength addition on the DSP. Using this algorithm, you can add two doublelength numbers (x,xx) and (y,yy) and represent the result as a doublelength number (z,zz).

The algorithm requires forming a doublelength number (r,rr) that represents an exact addition of x and y. Generating a second number, s = ((rr + yy) + xx), results in a number pair (r,s) that approximates the addition of (x,xx) and (y,yy). Finally, an exact addition of r and s generates a doublelength number (z,zz) that has the same value as (x,xx) + (y,yy).

To obtain exact results for addition and subtraction, subtraction and addition must be optimal; this is guaranteed by following each subtraction or addition instruction on the DSP with a round instruction.

```
; Calculate the doublelength sum of (x,xx) and (y,yy),
; the result being (z,zz)
;
        r = x + y;
        if (abs(x) > abs(y))
              s = x - r + y + yy + xx;
        else
              s = y - r + x + xx + yy;
        z = r + s;
        zz = r - z + s;
```

Figure 3. Doublelength Addition

Exact Singlelength Multiplication

The exact singlelength multiplication is shown in Figure 4. The algorithm requires breaking the x and y mantissas into half-length numbers, referred to as head (hx,hy) and tail (tx,ty) sections [2]. This algorithm requires addition and subtraction to be optimal and multiplication faithful. The TMS320C30 DSP multiplication result is faithful if the contents of the extended-precision register are truncated.

To split x and y into two half-length numbers, a constant value is needed that is dependent on the number of available digits. The TMS320C30 device has t = 24 bits of mantissa in the single-precision format. Equation (5) shows that head section hx is chosen to be as near to the value of x as possible.

$$hx = round(m(x)2^{-t1})2^{e(x)+t1} \tag{5}$$

Also, t1 is chosen to be approximately one-half of the available precision, or 12, on the processor. This effectively breaks the mantissa into half-length values. Equation (5) shows that hx is obtained by rounding and is defined to be an element of R{t1}. The tail section tx is easily obtained by subtracting hx from x. Since floating-point subtraction can be made optimal on the TMS320C30, it follows that tx is an element of R{t1 − 1}. Setting the constant equal to 2^{12} does not always satisfy Equation (5) when t is even. When the constant is set to $2^{12} + 1$, the definition of Equation (5) is satisfied. The proof for the above is given in Reference [2].

```
; Calculate the exact product of x and y, the result being
; a doublelength number (z,zz). This algorithm uses the
; following syntax when called from a user program as shown
; mult12 (x,y,z,zz);
;
        p = x × constant;
        hx = x − p + p;
        tx = x − hx;

        p = y × constant;
        hy = y − p + p;
        ty = y − hy;

        p = hx × hy;
        q = hx × ty + tx × hy;
        z = p + q;
        zz = p − z + q + tx × ty;
```

Figure 4. Exact Singlelength Product

Doublelength Multiplication

The doublelength multiplication algorithm, shown in Figure 5, relies on the singlelength algorithm discussed earlier. The algorithm generates a nearly doublelength approximation of the output result (c,cc). Note that the exact singlelength multiplication routine is used for this approximation. Exact addition is used to generate a doublelength floating-point number that is the closest approximation to the actual result.

The doublelength product program implementation uses the TMS320C30 stack capabilities to save some intermediate variables. These programs are written to be used as callable functions or macros in your program. In either case, the stack pointer must be set to a valid memory segment for proper code execution.

```
; Calculate the doublelength product of (x,xx) and (y,yy)
; the result being a nearly doublelength number (z,zz).
; Program uses exact singlelength multiplication, mult12 (.).
;
        mult12 (x, y, c, cc);
        cc = x × yy + xx × y + cc;
        z = c + cc;
        zz = c − z + cc;
```

Figure 5. Exact Doublelength Product

Doublelength Quotient and Square Root

Figures 6 and 7 show the algorithm used in calculating the doublelength quotient and doublelength square root routines. Singlelength multiplication is used to generate a doublelength approximation of the quotient or square root values. As with doublelength multiplication, exact addition is used to generate a doublelength floating-point result.

```
;
; Calculates the doublelength quotient of (x,xx) and (y,yy)
; the result being (z,zz)
;
        c = x / y;
        mult12(c, y, u, uu);
        cc = (x − u − uu + xx − c × yy) / y;
        z = c + cc;
        zz = c − z + cc;
```

Figure 6. Doublelength Quotient

```
; Calculate the doublelength square root of (x,xx), the
; result being (z,zz)
;
        if (x>0) {
                c = sqrt (x);
                mult12 (c, c, u, uu);
                cc = (x − u − uu + xx) × 0.5 / c;
                z = c + cc;
                zz = c − z + cc;}
        else {
                z = zz = 0.};
```

Figure 7. Doublelength Square Root

Error Analysis

This section discusses and determines an upper bound for the error generated in forming a doublelength result. The value of the doublelength number (z,zz) is equal to z + zz. Singlelength addition, subtraction, and multiplication results are always exact. In doublelength addition, any error introduced in the end result is generated by calculating the zz term. An upper bound error magnitude has been calculated in Reference [2] and is shown in Equation (6) as follows:

$$|E^{+}| \leq \{|x+xx| + |y+yy|\} \times 2^{2-2t} = |Z| \times 2^{2-2t} \quad (6)$$

where t = 24 for this system. This gives an upper bound of $|Z| \times 2^{-46}$, or approximately $|Z| \times 1.42 \times 10^{-14}$. This translates to a theorical accuracy greater than 13 decimal places. Table 1 shows an example of doublelength addition using the exact addition algorithm previously described. The numbers in the left column represent TMS320C30 hexadecimal notation for the floating-point results, and (z,zz) is the decimal equivalent of the doublelength output result. Appendix B shows a listing of C programs (exact) that convert from TMS320C30 hexadecimal notation to decimal notation.

Table 1. Exact Singlelength Arithmetic Examples

Singlelength Addition		
x = 217FFFFFh		
y = 0C7FFFFFh		
z = 22000003h	(z,zz) =	17179876351.9995117 (Exact)
zz = 097FFFF8h		17179876351.9995117 (DSP)
x = FC7C8923h		
y = 0A29A7E5h		
z = 0A29ABD8h	(z,zz) =	1357.37010409682989 (Exact)
zz = EFA46000h		1357.37010409682989 (DSP)
Singlelength Multiplication		
x = 0F7FFFFFh		
y = 21FFFFFFh		
z = 30800000h	(z,zz) =	−562949986975740 (Exact)
zz = 18800002h		−562949986975740 (DSP)
x = FC7CB923h		
y = 0A29A7E5h		
z = 07277BF7h	(z,zz) =	167.484236862815123 (Exact)
zz = EBA714F0h		167.484236862815123 (DSP)

The doublelength product, quotient, and square-root algorithms all have a small relative error. The upperbound error magnitude for each is given in Equations (7) through (9).

$$|E^{\times}| = (|x + xx| \times |y + yy|) \times 11 \times 2^{-48} \tag{7}$$

$$|E^{\div}| = (|x + xx| \div |y \times yy|) \times 21.1 \times 2^{-48} \tag{8}$$

$$|E^{\surd}| = \mathrm{sqrt}(|x + xx|) \times 12.7 \times 2^{-48} \tag{9}$$

Equation (7) establishes an upperbound of $|Z| \times 3.9 \times 10^{-14}$, or approximately 13 decimal digits of accuracy for doublelength multiplication. Similarly, an upperbound of $|Z| \times 7.5 \times 10^{-14}$, or greater than 13 decimal digits for the doublelength square-root algorithm, is established. Table 2 shows examples for each algorithm discussed, along with the algorithm output and expected theorical output.

Table 2. Exact Doublelength Arithmetic Examples

Doublelength Multiplication		
x = 22000000h		
xx = 097FFFFEh		
y = 21000001h		
yy = 097FFFFEh		
z = 43000002h	(z,zz) =	1.47573996570139475 × 10^{20} (Exact)
zz = 2A7FFFFCh		1.47573996570139427 × 10^{20} (DSP)
x = 22000003h		
xx = 097FFFF8h		
y = 0A29ABD8h		
yy = EFA46000h		
z = 2C29ABDDh	(z,zz) =	23319450552284.2434 (Exact)
zz = 13907DC2h		23319450552284.1250 (DSP)

Doublelength Quotient		
x = 43000002h		
xx = 2A7FFFFCh		
y = 2C29ABDDh		
yy = 13907DC2h		
z = 1641205Ah	(z,zz) =	6328365.08044074177 (Exact)
zz = FC24BE20h		6328365.08044075966 (DSP)
x = 22000000h		
xx = 097FFFFEh		
y = 21000001h		
yy = 097FFFFEh		
z = 007FFFFDh	(z,zz) =	1.99999964237223082 (Exact)
zz = D3400000h		1.99999964237217398 (DSP)

Doublelength Square Root		
x = 2C2BDD00h		
xx = 3907DC2h		
z = 61451A4h	(z,zz) =	4860114.04539400958 (Exact)
zz = FB39EF11h		4860114.04539400712 (DSP)
x = 21000001h		
xx = 097FFFFEh		
z = 103504F5h	(z,zz) =	92681.9110722252960 (Exact)
zz = F7BC0784h		92681.9110722253099 (DSP)

Note that the results were obtained using the programs shown in Appendix B. The C programs were created and compiled on a 80386-based microcomputer running under MS-DOS 3.3.

How to Generate C-Callable Functions

The source listings for the extended-precision arithmetic presented in Appendix A are optimized for execution speed and code size. These routines are designed to be used as macros in a user program environment or, with a few adjustments, as a C function.

This section provides an overview of TMS320C30 C compiler calling conventions necessary to create functions that can be added to the C compiler library. You need a working knowledge of C language to understand the terminology in this section [4, 5, 6].

The C compiler uses the processor stack to pass arguments to functions, store local variables, and save temporary values. The C compiler uses two registers of the TMS320C30 to manage the stack pointer (SP) and the frame pointer (AR3).

When a C program calls a function, it must

1. Push the arguments onto the stack,
2. Call the function, and
3. Pop the arguments off the stack,

in that order.

On the other hand, the called C function must perform the following tasks:

1. Set up a local frame by saving the old frame pointer on the stack.
2. Assign the new frame pointer to the current value of stack pointer.
3. Allocate the frame.
4. Save any dedicated registers that the function modifies.
5. Execute function code.
6. Store a scalar value in R0.
7. Deallocate the frame.
8. Lastly, restore the old frame pointer [4].

The following code segment shows the singlelength addition routine modified to be in C-callable form. Note that registers R4 through R7 and AR4 through AR7 are dedicated registers used by the compiler. These registers must be saved as floating-point values.

```
single    .set    0FFh
fp        .set    ar3
x         .set    r0
y         .set    r1
z         .set    r2
zz        .set    r3
```

```
w       .set    r4
x1      .set    r2
y1      .set    r3
        .global __add12:
        .width  96
        .text
__add12:
        push    fp              ; Save old fp
        pushf   r4
        push    r4
        ldi     sp,fp           ; Point to top of stack
        ldi     *-fp[2],r0      ; Load x into r0
        ldi     *-fp[3],r1      ; Load y into r1
        absf    x,x1
        absf    y,y1
        cmpf    y1,x1           ; |x| > |y|
        ldflt   x,x1
        ldflt   y,x
        dflt    x1,y
;
        addf3   x,y,z           ; z = x + y
        rnd     z
        subf3   x,z,w           ; Form w = z - x
        rnd     w
        subf3   w,y,zz          ; zz = y - (y - w)
        rnd     zz
        pop     r4
        popf    r4
        pop     fp              ; Restore fp
        retsu
        .end
```

Conclusion

This report presented an implementation of extended-precision arithmetic routines for the TMS320C30 DSP. The programs presented include singlelength floating-point addition, subtraction, and multiplication, which produce exact doublelength results. Doublelength floating-point addition, subtraction, multiplication, division, and square root were also presented. The doublelength floating-point routines all had a small relative error that appeared in the correction term zz. However, it has been shown that the accuracy of the doublelength floating-point result is at least 13 decimal digits. Table 3 is a summary of information about the routines contained in Appendices A and B. Execution times shown

in the table are given only for the routines in Appendix A. These times do not include the call and return if the routine is implemented as a called function. They also do not include any context saves and restores that may be required.

Table 3. Summary Information

Routine	Mnemonic	Appendix	Code Size (Words)	Execution (Cycles)
Singlelength Add	_add12	A1	12	12
Doublelength Add	_dbladd	A2	25	25
Singlelength Multiply	_mult12	A3	35	35
Doublelength Multiply	_mult2	A4	51	51
Doublelength Divide	_div2	A5	115	115
Doublelength Square Root	_sqrt2	A6	163	163
Change Two Single-Precision TMS320C30 Numbers to One Double-Precision Result	C30DBL	B1		
Change Two Double-Precision TMS320C30 Numbers to a Double-Precision Result	C30DBL2	B2		

References

[1.] *Third-Generation TMS320 User's Guide* (literature number SPRU031), Texas Instruments, Inc., 1988.

[2.] Dekker, T.J.,"A Floating-Point Technique for Extending the Available Precision", Numer. Math. 18, 1971, pp 224-242.

[3.] Linnainmaa, S.,"Software for Doubled-Precision Floating-Point Computations", *ACM Transactions on Mathematical Software*, Vol. 7, No. 3, Sept. 1981, pp 272-283.

[4.] *TMS320C30 C Compiler* (literature number SPRU034), Texas Instruments, Inc., 1988.

[5.] Kernigan, B.W. and Ritchie, D.M., *The C Programming Language*, 2nd Revision, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[6.] Kochan, S.G., *Programming in C*, Second Edition, Howard K. Sams, Indianapolis, Indiana, 1988.

Appendix A

Appendix A1. Single Length Add

```
*****************************************************
*  FUNCTION DEF : _add12
*
*  AUTHOR: Al Levrich 2/21/89
*    Texas Instruments, Inc.
*
*  Entry Conditions:
*  Upon entry (r0,r1) contains (x,y)
*  Exit Conditions:
*  Upon exit (r2,r3) contains (z,zz).
*  Registers Affected:
*  r0, r1, r2, r3, r4
*
*  Revision: Original
*  Execution Time: 12 cycles
*****************************************************
single  .set    0ffh
        .global _add12
x       .set    r0
y       .set    r1
z       .set    r2
zz      .set    r3
w       .set    r4
x1      .set    r2
y1      .set    r3
        .text
_add12:
        absf    x,x1
        absf    y,y1
        cmpf    y1,x1           ; |x1 > |y1 ?
        ldflt   x,x1            ; if not, exchange x & y
        ldflt   y,x
        ldflt   x1,y
*
        addf3   x,y,z           ; z = x + y
        rnd     z
        subf3   x,z,w           ; form w = z - x
        rnd     w
subf3   w,y,zz          ; zz = y - w
rnd     zz
retsu
.end
```

Appendix A2. Double Length Add

```
*********************************************************
* FUNCTION DEF : dbladd
*
* AUTHOR: Al Lovrich   2/21/89
*         Texas Instruments, Inc.
*
* Entry Conditions:
*      Upon entry (r0,r1) contains (x,xx) and
*      (r2,r3) contain (y,yy).
* Exit Conditions:
*      Upon exit (r4,r5) contains (z,zz).
* Registers Affected:
*      r0, r1, r2, r3, r4, r5, r6, r7
*
* Revision: Original
* Execution time:  25 cycles
*********************************************************
        .global _dbladd
x       .set    r0
xx      .set    r1
y       .set    r2
yy      .set    r3
z       .set    r4
zz      .set    r5
x1      .set    r6
y1      .set    r7
r       .set    r6
s       .set    r7

        .text
_dbladd:
       absf    x,x1
       absf    y,y1
       cmpf    y1,x1                   ; check for |x| > |y|
       ldflt   x,x1                    ; if not, exchange (x,xx)
       ldflt   xx,y1                   ; and (y,yy)
       ldflt   y,x
       ldflt   yy,xx
       ldflt   x1,y
       ldflt   y1,yy
;
       addf3   x,y,r                   ; r = x + y
       rnd     r
*
       subf3   r,x,s                   ; s = x - r
       rnd     s
       addf3   y,s,s                   ; s = x - r + y
       rnd     s
       addf    yy,s                    ; s = x - r + y + yy
       rnd     s
       addf    xx,s                    ; s = x - r + y + yy + xx
       rnd     s
*
       addf3   s,r,z                   ; z = r + s
       rnd     z
*
       subf3   z,r,zz                  ; zz = r - z
       rnd     zz
       addf3   s,zz,zz                 ; zz = r - z + s
       rnd     zz
       retsu
        .end
```

```
*********************************************************
*  FUNCTION DEF : _mult12
*
*  AUTHOR: Al Lovrich  2/21/89
*    Texas Instruments, Inc.
*
*  Entry Conditions:
*       Upon entry (r0,r1) contains (x,y)
*  Exit Conditions:
*       Upon exit  (r0,r1) contains (z,zz).
*  Registers Affected:
*       r0, r1, r2, r3, r4, r5, r6, r7
*
*  Revision: Original
*  Execution Time: 35 Cycles
*********************************************************
          .global   _mult12
single    .set      0ffh
x         .set      r0
y         .set      r1
p         .set      r2
x         .set      r3
tx        .set      r4
q         .set      r5
hy        .set      r5
ty        .set      r6
z         .set      r0
zz        .set      r1
.temp     .set      r7
          .text
_mult12:
          ldf       @constant,temp
          mpyf3     temp,x,p        ; p = x * constant
          andn      single,p        ; fl(*) is faithful
*
          subf3     p,x,hx          ; hx = x - p
          rnd       hx

          addf3     hx,p,hx         ; hx = x - p + p
          rnd       hx
*
          subf3     hx,x,tx         ; tx = x - hx
          rnd       tx
*
          mpyf3     temp,y,p        ; p = y * constant
          andn      single,p        ; fl(*) is faithful
*
          subf3     p,y,hy          ; hy = y - p
          rnd       hy
          addf3     hy,p,hy         ; hy = y - p + p
          rnd       hy
*
          subf3     hy,y,ty         ; ty = y - hy
          rnd       ty
*
          mpyf3     hx,hy,p         ; p = hx * hy
          andn      single,p        ; fl(*) is faithful
*
          mpyf3     hx,ty,temp      ; temp = hx * ty
          andn      single,temp     ; fl(*) is faithful
          mpyf3     tx,hy,q         ; q = tx * hy
          andn      single,q        ; fl(*) is faithful
          addf3     q,temp,q        ; q = hx * ty + tx * hy
          rnd       q
*
          addf3     p,q,z           ; z = p + q
          rnd       z
*
          subf3     z,p,zz          ; zz = p - z
          rnd       zz
          addf      q,zz            ; zz = p - z + q
          rnd       zz
          mpyf3     tx,ty,temp      ; temp = tx * ty
          andn      single,temp     ; fl(*) is faithful
          addf3     zz,temp,zz      ; zz = p - z + q + tx * ty
          rnd       zz
*
          retsu
          .data

constant:
          .float    4097            ; constant = 2^(24-24/2)+1
          .end
```

```
*******************************************************
*   FUNCTION DEF : _mult2
*
*   AUTHOR: Al Lovrich    2/21/89
*           Texas Instruments, Inc.
*   Entry Conditions:
*           Upon entry (r0,r1) contains (x,y),
*           and (r2,r3) contains (xx,yy).
*   Exit Conditions:
*           Upon exit  (r0,r1) contains (z,zz).
*   Registers Affected:
*           r0, r1, r2, r3, r4, r5, r6, r7
*
*   Algorithm used:
*           mult12(x, y, c, cc);
*           cc = x * yy + xx * y + cc;
*           z  = c + cc;
*           zz = c - z + cc;
*
*   Revision: Original
*   Execution Time: 51 cycles
*******************************************************
            .global     _mult2
single      .set        0ffh
x           .set        r0
y           .set        r1
p           .set        r2
hx          .set        r3
tx          .set        r4
q           .set        r5
hy          .set        r5
ty          .set        r6
z           .set        r0
zz          .set        r1
xx          .set        r2
yy          .set        r3
c           .set        r4
cc          .set        r6
temp0       .set        r6
temp        .set        r7
            .text
_mult2:
            mpyf3       x,yy,temp0      ; temp0 = x*yy
            andn        single,temp0
            mpyf3       y,xx,temp       ; temp = y*xx
            andn        single,temp
            addf        temp0,temp      ; temp = x*yy + y*xx
            rnd         temp
            pushf       temp            ; (x*yy + y*xx)
*
*  mult12(x, y, c, cc)
*
            ldf         @constant,temp
            mpyf3       temp,x,p        ; p = x * constant
            andn        single,p
*
            subf3       p,x,hx          ; hx = x - p
            rnd         hx
            addf3       hx,p,hx         ; hx = x - p + p
            rnd         hx
*
            subf3       hx,x,tx         ; tx = x - hx
            rnd         tx
*
            mpyf3       temp,y,p        ; p = y * constant
            andn        single,p
*
            subf3       p,y,hy          ; hy = y - p
            rnd         hy
            addf3       hy,p,hy         ; hy = y - p + p
            rnd         hy
*
            subf3       hy,y,ty         ; ty = y - hy
            rnd         ty
*
            mpyf3       hx,hy,p         ; p = hx * hy
            andn        single,p
*
            mpyf3       hx,ty,temp      ; temp = hx * ty
            andn        single,temp
            mpyf3       tx,hy,q         ; q = tx * hy
            andn        single,q
            addf3       q,temp,q        ; q = hx * ty + tx * hy
            rnd         q
*
            mpyf3       tx,ty,temp      ; temp = tx * ty
            andn        single,temp
            addf3       p,q,c           ; c = p + q
            rnd         c
*
            subf3       c,p,cc          ; cc = p - c
            rnd         cc
            addf        q,cc            ; cc = p - c + q
            rnd         cc
            addf        temp,cc         ; cc = p - c + q + tx * ty
            rnd         cc
*
*  restore variables
*
break:
            popf        temp            ; x*yy + y*xx
*
*  cc = x * yy + xx * y + cc
*
            addf        temp,cc         ; cc = x * yy + xx * y + cc
            rnd         cc
*
*  z = c + cc
*
            addf3       cc,c,z          ; z = c + cc
            rnd         z
*
*  zz = c - z + cc
*
            subf3       z,c,zz          ; zz = c - z
            rnd         zz
            addf3       zz,cc,zz        ; zz = c - z + cc
            rnd         zz
*
            retsu
            .data
constant:
            .float      4097            ; constant = 2^(24-24/2)+1
            .end
```

```
*********************************************************
*   FUNCTION DEF : _div2
*
*   AUTHOR: Al Lovrich 2/21/89
*     Texas Instruments, Inc.
*
*   Entry Conditions:
*   Upon entry (r0,r1) contains (x,y),
*   and (r2,r3) contains (xx,yy).
*   Exit Conditions:
*   Upon exit  (r0,r1) contains (z,zz).
*   Registers Affected:
*   r0, r1, r2, r3, r4, r5, r6, r7
*
*   Algorithm used:
*   c = x / y;
*   mult12(c, y, u, uu);
*   cc = ( x - u - uu + xx - c * yy ) / y;
*   z  = c + cc;
*   zz = c - z + cc;
*
*   Revision: Original
*   Execution Time: 115 Cycles
*********************************************************
          .global   _div2
single    .set      0ffh
x         .set      r0
y         .set      r1
p         .set      r2
hx        .set      r3
tx        .set      r4
yl        .set      r4
q         .set      r5
hy        .set      r5
ty        .set      r6
z         .set      r0
zz        .set      r1
xx        .set      r2
yy        .set      r3
temp      .set      r7
temp1     .set      r3
temp2     .set      r1
c         .set      r2
cc        .set      r3
u         .set      z
uu        .set      zz
          .text
*
_div2:
          pushf     yy              ; save yy
          pushf     xx              ; save xx
          pushf     x               ; save x
          pushf     y               ; save y
*
*  c = x / y;
*
*  The floating-point number v is stored in R1. After the computation is
*  completed, 1/v is also stored in R4.
*
*  Register used as input: R1
*  Registers modified: R0, R1, R2, R3
*  Register containing result: R4
*
inv_f:    ldf       r1,r3           ; v is saved for later.
          absf      r1              ; The algorithm uses v = |v|.
*
*  Extract the exponent of v.
*
          pushf     r1
          pop       r0
          ash       -24,r0          ; The 8 LSBs of R1 contain the exponent
*                                   ; of v.
*
*  A few comments on boundary conditions. If e = -128, then v = 0. The
*  following x[0] calculation yields R1 =  -128 - 1 = 127 and the algoritm
*  overflow and saturate since x[0] is large. This seems reasonable. If 127,
*  the R1 = -127 - 1 = -128. Thus x[0] = 0 and this will cause the algorithm
*  to yield zero. Since the mantissa of v is always between 1 this is also
*  reasonable. As a result, boundary conditions are handled automatically in
*  a reasonable fashion.
*
*  x[0] formation given the exponent of v.
*
          negi      r0
          subi      1,r0            ; Now we have -e-1, the exponent of x[0].
          ash       24,r0
          push      r0
          popf      r0              ; Now R1 = x[0] = 1.0 * 2**(-e-1).
*
*  Now the iterations begin.
*
          mpyf3     r0,r1,r2        ; R2 = v * x[0]
          andn      single,r2
          subrf     2.0,r2          ; R2 = 2.0 - v * x[0]
          rnd       r2
          mpyf      r2,r0           ; R1 = x[1] = x[0] * (2.0 - v * x[0])
          andn      single,r0
*
          mpyf      r0,r1,r2        ; R2 = v * x[1]
          andn      single,r2
          subrf     2.0,r2          ; R2 = 2.0 - v * x[1]
          rnd       r2
          mpyf      r2,r0           ; R1 = x[2] = x[1] * (2.0 - v * x[1])
          andn      single,r0
```

```
*
        mpyf    r0,r1,r2        ; R2 = v * x[2]
        andn    single,r2
        subr    2.0,r2          ; R2 = 2.0 - v * x[2]
        rnd     r2
        mpyf    r2,r0           ; R1 = x[3] = x[2] * (2.0 - v * x[2])
        andn    single,r0
*
        mpyf    r0,r1,r2        ; R2 = v * x[3]
        andn    single,r0
        subrf   2.0,r2          ; R2 = 2.0 - v * x[3]
        rnd     r2
        mpyf    r2,r0           ; R1 = x[4] = x[3] * (2.0 - v * x[3])
*
        andn    single,r0       ; This minimizes error in the LSBs.
*
*  For the last iteration we use the formulation:
*  x[5] = (x[4] * (1.0 - (v * x[4]))) + x[4]
*
        mpyf    r0,r1,r2        ; R2 = v * x[4] = 1.0..01.. => 1
        andn    single,r2
        subrf   1.0,r2          ; R2 = 1.0 - v*x[4] = 0.0..01... => 0
        rnd     r2
        mpyf    r0,r2           ; R2 = x[4] * (1.0 - v * x[4])
        andn    single,r2
        addf    r2,r0           ; R2 = x[5] = (x[4]*(1.0-(v*x[4])))+x[4]
*
        rnd     r0,r1           ; Round since this is follow by a MPYF.
*
*  Now the case of v < 0 is handled.
*
        negf    r1,r2
        ldf     r3,r3           ; This sets condition flags.
        ldfn    r2,r1           ; If v < 0, then R1 = -R1
*
        ldf     r1,r4           ; save 1/y
*
*  restore variables
*
        popf    y               ; restore y
        popf    x               ; restore x
        pushf   x               ; save x
*
        mpyf    y1,x            ; c = x * ( 1/y)
        andn    single,x
*
*  save variables
*
        pushf   x               ; save c
        pushf   y1              ; save 1/y
*
*  mult12(c, y, u, uu)
*
        ldf     @constant,temp
        mpyf3   temp,x,p        ; p = x * constant
        andn    single,p
*
        subf3   p,x,hx          ; hx = x - p
        rnd     hx
        addf3   hx,p,hx         ; hx = x - p + p
        rnd     hx
*
        subf3   hx,x,tx         ; tx = x - hx
        rnd     tx
*
        mpyf3   temp,y,p        ; p = y * constant
        andn    single,p
*
        subf3   p,y,hy          ; hy = y - p
        rnd     hy
        addf3   hy,p,hy         ; hy = y - p + p
        rnd     hy
*
        subf3   hy,y,ty         ; ty = y - hy
        rnd     ty
*
        mpyf3   hx,hy,p         ; p = hx * hy
        andn    single,p
*
        mpyf3   hx,ty,temp      ; temp = hx * ty
        andn    single,temp
        mpyf3   tx,hy,q         ; q = tx * hy
        andn    single,q
        addf3   q,temp,q        ; q = hx * ty + tx * hy
        rnd     q
*
*  perform tx * ty operation and store the result in temp. This is to
*  optimize use of registers on the device.
*
        mpyf3   tx,ty,temp      ; temp = tx * ty
        andn    single,temp
        addf3   p,q,u           ; u = p + q
        rnd     u
*
        subf3   u,p,uu          ; uu = p - u
        rnd     uu
        addf    q,uu            ; uu = p - u + q
        rnd     uu
```

```
            popf    y1          ; restore 1/y
            popf    c           ; restore c
            popf    temp        ; restore x
            subf3   u,temp,cc   ; cc = x - u
            rnd     cc
            subf    uu,cc       ; cc = x - u - uu
            rnd     cc
            popf    temp        ; restore xx
            addf    temp,cc     ; cc = x - u - uu + xx
            rnd     cc
            popf    temp        ; restore yy
            mpyf    c,temp      ; c * yy
            andn    single,temp
            subf    temp,cc     ; cc = x - u - uu + xx - c * yy
            rnd     cc
            mpyf    y1,cc       ; cc = ( x - u - uu + xx - c * yy ) / y
            andn    single,cc
*
*  z = c + cc
*
            addf3   c,cc,z      ; z = c + cc
            rnd     z
*
*  zz = c - z + cc
*
            subf    z,c,zz      ; zz = c - z
            rnd     zz
            addf    cc,zz       ; zz = c - z + cc
            rnd     zz
*
            retsu
            .data
constant:
            .float  4097        ; constant = 2^(24-24/2)+1
            .end
```

```
*****************************************************
*   FUNCTION DEF : _sqrt2
*
*   AUTHOR: Al Lovrich    2/21/89
*           Texas Instruments, Inc.
*
*   Entry Conditions:
*         Upon entry (r0,r1) contains (x,xx).
*   Exit Conditions:
*         Upon exit  (r0,r1) contains (z,zz).
*   Registers Affected:
*         r0, r1, r2, r3, r4, r5, r6, r7
*
*   Algorithm used:
*         c = sqrt(x);
*         mult12(c, c, u, uu);
*         cc = ( x - u - uu + xx ) * 0.5 / c;
*         z  = c + cc;
*         zz = c - z + cc;
*
*   Revision: Original
*   Execution Time: 163 Cycles
*****************************************************
            .global   _sqrt2
single      .set      0ffh
x           .set      r0
y           .set      r1
p           .set      r2
hx          .set      r3
tx          .set      r4
q           .set      r5
hy          .set      r6
ty          .set      r7
z           .set      r0
zz          .set      r1
xx          .set      r1
temp        .set      r7
c           .set      r2
cc          .set      r3
u           .set      z
uu          .set      zz
c1          .set      r0
            .text
_sqrt2:
*
*  c = sqrt(x)
*
*  Extract the exponent of v.
*
            ldf       r0,r3         ; save v
            retsle                  ; return if number non-positive
            pushf     xx            ; save xx
            pushf     x             ; save x
            mpyf      2.0,r0        ; add a rounding bit in the exponent
            andn      single,r0
            pushf     r0
            pop       r1
            ash       -25,r1        ; The 8 LSBs of R1 contain 1/2 the expon
*
*  x[0] formation given the exponent of v.
*
            negi      r1
            ash       24,r1
            push      r1
            popf      r1            ; Now r1 = x[0] = 1.0 * 2**(-e/2).
*
*  Generate v/2.
*
            mpyf      0.25,r0       ; v/2 and take rounding bit out.
            andn      single,r0
*
*  Now the iterations begin.
*
            mpyf      r1,r1,r2      ; r2 = x[0] * x[0]
            andn      single,r2
            mpyf      r0,r2         ; r2 = (v/2) * x[0] * x[0]
            andn      single,r2
            subrf     1.5,r2        ; r2 = 1.5 - (v/2) * x[0] * x[0]
            rnd       r2
            mpyf      r2,r1         ; r1 = x[1] = x[0] * (1.5 - (v/2)*x[0]*x
            andn      single,r1
*
            mpyf      r1,r1,r2      ; r2 = x[1] * x[1]
            andn      single,r2
            mpyf      r0,r2         ; r2 = (v/2) * x[1] * x[1]
            andn      single,r2
            subrf     1.5,r2        ; r2 = 1.5 - (v/2) * x[1] * x[1]
            rnd       r2
            mpyf      r2,r1         ; r1 = x[2] = x[1] * (1.5 - (v/2)*x[1]*x
            andn      single,r1
*
            mpyf      r1,r1,r2      ; r2 = x[2] * x[2]
            andn      single,r2
            mpyf      r0,r2         ; r2 = (v/2) * x[2] * x[2]
            andn      single,r2
            subrf     1.5,r2        ; r2 = 1.5 - (v/2) * x[2] * x[2]
            rnd       r2
            mpyf      r2,r1         ; r1 = x[3] = x[2] * (1.5 - (v/2)*x[2]*x
            andn      single,r1
*
            mpyf      r1,r1,r2      ; r2 = x[3] * x[3]
            andn      single,r2
            mpyf      r0,r2         ; r2 = (v/2) * x[3] * x[3]
            andn      single,r2
```

```
        subrf   1.5,r2      ; r2 = 1.5 - (v/2) * x[3] * x[3]
        rnd     r2
        mpyf    r2,r1       ; r1 = x[4] = x[3] * (1.5 - (v/2)*x[3]*x
        andn    single,r1
*
        mpyf    r1,r1,r2    ; r2 = x[4] * x[4]
        andn    single,r2
        mpyf    r0,r2       ; r2 = (v/2) * x[4] * x[4]
        andn    single,r2
        subrf   1.5,r2      ; r2 = 1.5 - (v/2) * x[4] * x[4]
        rnd     r2
        mpyf    r2,r1       ; r1 = x[5] = x[4] * (1.5 - (v/2)*x[4]*x
*
        andn    single,r1
        ldf     r1,r0
*
        mpyf    r3,r0       ; sqrt(v) from sqrt(v**(-1))
        andn    single,r0
*
*  Save variables
*
        pushf   x           ; save c = sqrt(x)
        ldf     x,y         ; get ready for multiplication
*
*  multl2(c, c, u, uu)
*
        ldf     @constant,temp
        mpyf3   temp,x,p    ; p = x * constant
        andn    single,p
*
        subf3   p,x,hx      ; hx = x - p
        rnd     hx
        addf    p,hx        ; hx = x - p + p
        rnd     hx
*
        subf3   hx,x,tx     ; tx = x - hx
        rnd     tx
*
        mpyf3   temp,y,p    ; p = y * constant
        andn    single,p
*
        subf3   p,y,hy      ; hy = y - p
        rnd     hy
        addf3   hy,p,hy     ; hy = y - p + p
        rnd     hy
*
        subf3   hy,y,ty     ; ty = y - hy
        rnd     ty
*
        mpyf3   hx,hy,p     ; p = hx * hy
        andn    single,p
*
        mpyf3   hx,ty,temp  ; temp = hx * ty
        andn    single,temp
        mpyf3   tx,hy,q     ; q = tx * hy
        andn    single,q
        addf    temp,q      ; q = hx * ty + tx * hy
        rnd     q
*
*  perform tx * ty operation and store the result in temp.
*  This is to optimize use of registers on the device.
*
        mpyf3   tx,ty,temp  ; temp = tx * ty
        andn    single,temp
        addf3   p,q,u       ; u = p + q
        rnd     u
*
        subf3   u,p,uu      ; uu = p - u
        rnd     uu
        addf    q,uu        ; uu = p - u + q
        rnd     uu
        addf    temp,uu     ; uu = p - u + q + tx * ty
        rnd     uu
*
*  cc = ( x - u - uu + xx ) * 0.5 / c
*
        popf    c           ; restore c
        popf    temp        ; restore x
        subf3   u,temp,cc   ; cc = x - u
        rnd     cc
        subf    uu,cc       ; cc = x - u - uu
        rnd     cc
        popf    temp        ; restore xx
        addf    temp,cc     ; cc = x - u - uu + xx
        rnd     cc
*
        pushf   cc          ; save cc
        pushf   c           ; save c
*
*  The floating-point number v is stored in R1. After the computation is
*  completed, 1/v is also stored in R4.
*
*  Register used as input: R2
*  Registers modified: R0, R1, R2, R3
*  Register containing result: R2
*
        ldf     r2,r3       ; v is saved for later.
        absf    r2          ; The algorithm uses v = |v|.
*
```

```
*  Extract the exponent of v.
*
          pushf     r2
          pop       r1
          ash       -24,r1          ; The 8 LSBs of R0 contain the exponent
*                                   ; of v
*
*  x[0] formation given the exponent of v.
*
          negi      r1
          subi      1,r1            ; Now we have -e-1, the exponent of x[0]
          ash       24,r1
          push      r1
          popf      r1              ; Now R0 = x[0] = 1.0 * 2**(-e-1).
*
*  Now the iterations begin.
*
          mpyf3     r1,r2,r0        ; R1 = v * x[0]
          andn      single,r0
          subrf     2.0,r0          ; R1 = 2.0 - v * x[0]
          rnd       r0
          mpyf      r0,r1           ; R0 = x[1] = x[0] * (2.0 - v * x[0])
          andn      single,r1
*
          mpyf      r1,r2,r0        ; R1 = v * x[1]
          andn      single,r0
          subrf     2.0,r0          ; R1 = 2.0 - v * x[1]
          rnd       r0
          mpyf      r0,r1           ; R0 = x[2] = x[1] * (2.0 - v * x[1])
          andn      single,r1
*
          mpyf      r1,r2,r0        ; R1 = v * x[2]
          andn      single,r0
          subrf     2.0,r0          ; R1 = 2.0 - v * x[2]
          rnd       r0
          mpyf      r0,r1           ; R0 = x[3] = x[2] * (2.0 - v * x[2])
          andn      single,r1
*
          mpyf      r1,r2,r0        ; R1 = v * x[3]
          andn      single,r0
          subrf     2.0,r0          ; R1 = 2.0 - v * x[3]
          rnd       r0
          mpyf      r0,r1           ; R0 = x[4] = x[3] * (2.0 - v * x[3])
*
          andn      single,r1
*
*  For the last iteration we use the formulation:
*  x[5] = (x[4] * (1.0 - (v * x[4]))) + x[4]
*
          mpyf      r1,r2,r0        ; R1 = v * x[4] = 1.0..01.. => 1
          andn      single,r0
          subrf     1.0,r0          ; R1 = 1.0 - v * x[4] = 0.0..01... => 0
          rnd       r0
          mpyf      r1,r0           ; R1 = x[4] * (1.0 - v * x[4])
          andn      single,r0
          addf      r0,r1           ; R0 = x[5] = (x[4]*(1.0-(v*x[4])))+x[4]
*
          rnd       r1,r2           ; Round since this is followed by a MPYF
*
*  Now the case of v < 0 is handled.
*
          negf      r2,r0
          ldf       r3,r3           ; This sets condition flags.
          ldfn      r0,r2           ; If v < 0, then R2 = -R2
*
*  restore variables
*
          popf      temp            ; restore c
          popf      cc              ; restore cc
          mpyf      0.5,cc          ; cc = ( x - u - uu + xx ) * 0.5
          andn      single,cc
          mpyf      r2,cc           ; cc = ( x - u - uu + xx ) * 0.5 / c
          andn      single,cc
*
*  z = c + cc
*
          addf3     temp,cc,z       ; z = c + cc
          rnd       z
*
*  zz = c - z + cc
*
          subf      z,temp,zz       ; zz = c - z
          rnd       zz
          addf      cc,zz           ; zz = c - z + cc
          rnd       zz
*
          retsu
          .data
constant:
          .float    4097            ; constant = 2^(24-24/2)+1
          .end
```

Appendix B

Appendix B1. Change Two Single-Precision Numbers to One Double-Precision Result

```
/* C30DBL — Program to operate on two single-precision numbers
   in C30 format and produce a double-precision result */
#include <math.h>
#include <stdio.h>

main()
{
 long double x, y, z;
 long int xi, yi;
 int  i, operation;
 long int c30toe(long int);

 i=1;
 do{
  printf("Type two C30 hex numbers\n");
  printf("x = ");
  scanf("%X",&xi);
  printf("y = ");
  scanf("%X",&yi);
  xi = c30toe(xi);
  x = (long double)*(float *)(&xi);
  yi = c30toe(yi);
  y = (long double)*(float *)(&yi);
  do{
   printf("Add(1), Sub(2), Mpy(3), Div(4), Sqrt(5)= ");
   scanf("%d", &operation);
  } while (operation<1 || operation>5);

  if (operation == 1) z = x + y;
  if (operation == 2) z = x - y;
  if (operation == 3) z = x * y;
  if (operation == 4) z = x / y;
  if (operation == 5) z = sqrt(x);
  printf("\nz = %.18Lg", z);

  printf("\n\nType in C30 hex result\n");
  printf("z = ");
  scanf("%X",&xi);
  printf("zz = ");
  scanf("%X",&yi);
  xi = c30toe(xi);
  x = (long double)*(float *)(&xi);
  yi = c30toe(yi);
  y = (long double)*(float *)(&yi);
  z = x - y;
  printf("\nz = %.18Lg", z);
  printf("\n\nType 0 to exit, else continue : ");
  scanf ("%d", &i);
 } while (i != 0);
}

/* C30TOE — routine to convert from a C30 floating point number to a
   number in ieee format.  Both input and output in hex. */

long int c30toe(long int x)
{
 long int mantissa, sign;
 long int exp;

 sign = x & 0x00800000;
 exp = x >> 24;

 /* exp=-128 corresponds to 0.  exp=-127 is denormalized in ieee:
    represent it as 0. */

 if (exp <= -127) return(0);

 /* add implied bit and sign-extend mantissa */

 mantissa = x & 0x007fffff;
 if (sign)
  mantissa |= 0xff000000;
 else
  mantissa |= 0x00800000;

 /* convert mantissa to sign-magnitude */

 if (sign) mantissa = -mantissa;

 /* adjust mantissa if it was -2.0 */

 if (mantissa == 0x01000000){
  exp++;
  mantissa = 0x00800000;
 }
 if (exp > 127) return(0);  /* too large number; return error */

 /* make exponent 127-excess and return ieee number */

 exp += 127;
 mantissa = (mantissa & 0x007fffff) | (sign << 8) | (exp << 23);

 return(mantissa);
}
```

Appendix B2. Change Two Double-Precision Numbers to One Double-Precision Result

```
/* C30DBL2  — Program to operate on two double-precision numbers
  in C30 format and produce a double-precision result */
#include <math.h>
#include <stdio.h>

main()
{
 long double x, y, z;
 long int x1, y1, xx1, yy1;
 int  i, operation;
 long int c30toe(long int);

i=1;
do{
 printf("Type two C30 hex numbers:\n");
 printf("x = ");
 scanf("%X",&x1);
 printf("xx = ");
 scanf("%X",&xx1);
 printf("y = ");
 scanf("%X",&y1);
 printf("yy = ");
 scanf("%X",&yy1);
 x1 = c30toe(x1);
 xx1 = c30toe(xx1);
 y1 = c30toe(y1);
 yy1 = c30toe(yy1);
 x = (long double)(*(float *)(&x1)) +
   (long double)(*(float *)(&xx1));
 y = (long double)(*(float *)(&y1)) +
   (long double)(*(float *)(&yy1));
 do{
  printf("Add(1), Sub(2), Mpy(3), Div(4), Sqrt(5): ");
  scanf("%d", &operation);
 } while (operation<1 || operation>5);

 if (operation == 1) z = x + y;
 if (operation == 2) z = x - y;
 if (operation == 3) z = x * y;
 if (operation == 4) z = x / y;
 if (operation == 5) z = sqrt(x);
 printf("\nz = %.18Lg", z);

 printf("\n\nType in C30 hex result:\n");
 printf("z = ");
 scanf("%X",&x1);
 printf("zz = ");
 scanf("%X",&y1);
 x1 = c30toe(x1);
 x = (long double)(*(float *)(&x1));
 y1 = c30toe(y1);
 y = (long double)(*(float *)(&y1));
 z = x + y;
  printf("\nz = %.18Lg", z);
  printf("\n\nType 0 to exit, else continue : ");
  scanf ("%d", &i);
 } while (i != 0);
}
/* C30TOE — routine to convert from a c30 floating point number to a
  number in ieee format.  Both input and output in hex. */

long int c30toe(long int x)
{
 long int mantissa, sign;
 long int exp;

 sign = x & 0x00800000;
 exp = x >> 24;

/* exp=-128 corresponds to 0.  exp=-127 is denormalized in ieee!
 represent it as 0. */

 if (exp <= -127) return(0);

/* add implied bit and sign-extend mantissa */

 mantissa = x & 0x007fffff;
 if (sign)
  mantissa |= 0xff000000;
 else
  mantissa |= 0x00800000;

/* convert mantissa to sign-magnitude */

 if (sign) mantissa = -mantissa;

/* adjust mantissa if it was -2.0 */

 if (mantissa == 0x01000000){
  exp++;
  mantissa = 0x00800000;
 }
 if (exp > 127) return(0);  /* too large number; return error */

/* make exponent 127-excess and return ieee number */

 exp += 127;
 mantissa = (mantissa & 0x007fffff) | (sign << 8) | (exp << 23);

 return(mantissa);
}
```